Compiling C Code 16

From the same set of source files, you can create radically different programs, depending on how you compile those files. This chapter describes how to compile ANSI-conformant code, choose how strictly THINK C typechecks your code, and produce programs that are tuned for a specific processor.

Other chapters in this manual give more information on how THINK C can compile your files:

- To optimize your code, see Chapter 17, “Optimizing Code.”
- To compile code written for other compilers, see Chapter 19, “Porting Code.”

Contents

Compiling ANSI-Conformant Code 329

Compiling C++-Compatible Code 330

Type Checking 331
Checking pointer types 332
Enforcing prototype use 333

Compiling Processor-Specific Code 335
Compiling code for the MC68020 335
Compiling code for the MC68881 336

Compiling Code to Debug It 336
Always generate stack frames 337
Generate profiler calls 338
MacsBug Names 338

Using the THINK C Project Prefix 338

Compiling ANSI-Conformant Code

For more information on how THINK C complies with the ANSI standard, see Chapter 29, "Language Reference."

To compile ANSI-conformant code, you have to change some settings in the Languages Settings page of the **THINK C options** dialog, shown in Figure 16-1. These settings turn on ANSI features and turn off the THINK C extensions.

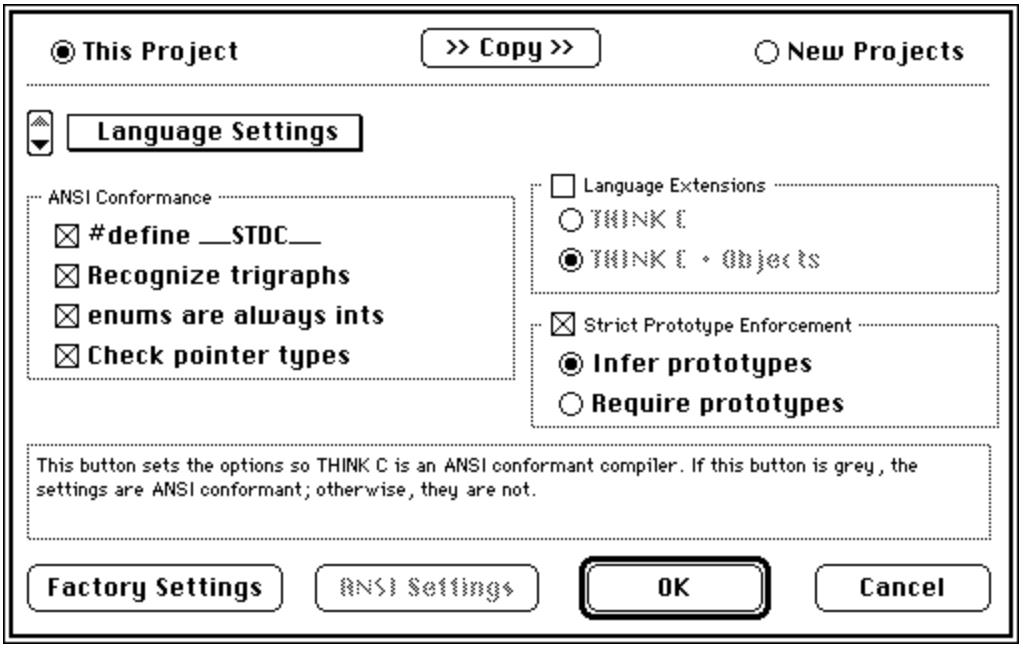

Figure 16-1 ANSI-conformant settings

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" and "Language extension options" in Chapter 15.

The ANSI Settings button on this page sets all the options on this page to their ANSI-conforming settings, as shown in this table:

ANSI Settings sets this…	**To this…**
#define __STDC__	On
Recognize trigraphs	On
enums are always ints	On
Check pointer types	On
Language Extensions	Off
Strict Prototype Enforcement	On and Infer prototypes option selected

Table 16-1 Effect of clicking the ANSI Settings button

This table explains what the options do when they have their ANSI-compatible settings:

The ANSI-conformant setting of option…	Does this…
#define __STDC__	#Defines `__STDC__`. If THINK C is ANSI-conformant, `__STDC__` is `1`; otherwise, `__STDC__` is `0`. If this option is off, THINK C doesn't define `__STDC__`.
Recognize trigraphs	Recognizes trigraphs. For example, `??'` is recognized as `^`. See "Disabling trigraphs" in Chapter 16.
enums are always ints	Makes enumerated constants the same size as an `int`. See "enums of any size" in Chapter 15.
Check pointer types	Checks whether pointers match when you assign pointers or perform pointer arithmetic. See "Checking pointer types" later in this chapter.
Language Extensions	Disables all THINK C extensions. See "THINK C Extensions" in Chapter 15.
Strict Prototype Enforcement	Infers a prototype when you first use a function. See "Enforcing prototype use" later in this chapter.

Table 16-2 ANSI-conformant option settings

Compiling C++-Compatible Code

For more information on the 4-byte ints option, see "Integer Representation" in Chapter 15. For more information on Native floating-point representation, see "Floating-Point Representation" in Chapter 15.

If your project contains both C and C++ source files, you have to change some settings in the Compiler Settings page of the **THINK C options** dialog. Since Symantec C++ does not support 2-byte integers and the universal floating-point format, you need to turn on

the 4-byte ints and Native floating-point format options, as in Figure 16-2. You don't need to change the settings of any other options.

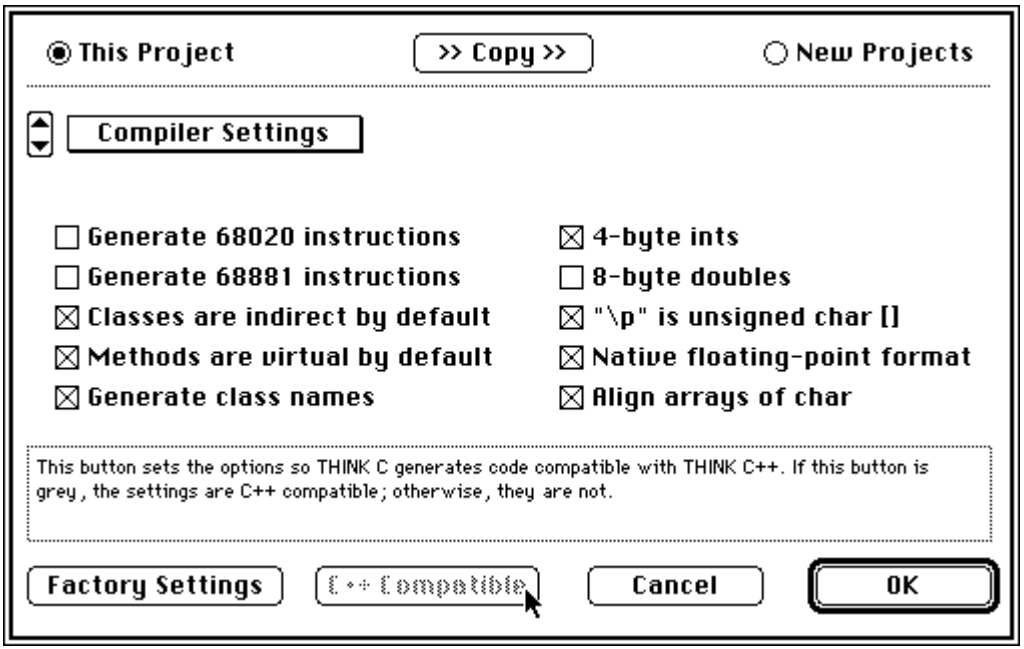

Figure 16-2 C++-compatible settings

The C++ Compatible button at the bottom of the Compiler Settings page turns on the 4-byte ints and Native floating-point format options. If those options are on, the button is grayed out.

Type Checking

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" and "Type-checking options" in Chapter 15.

THINK C lets you choose how strictly it enforces type checking. You can have THINK C make sure that pointer types are assignment-compatible, and you can enforce strict type checking by requiring a function prototype for every function. These options are on the Language Settings page of the **THINK C options** dialog.

If you want to compile ANSI-conformant code, you must set these options a certain way. For more information, see "Compiling ANSI-Conformant Code" earlier in this chapter.

Checking pointer types

When the Check pointer types option is on, THINK C makes sure that pointer types match when you assign one pointer to another or when you do pointer arithmetic. If this option is off, THINK C treats all pointers as equivalent types, and it won't display the `pointer types do not match` error message. When subtracting two pointers, however, the two types must be pointers to objects of the same size.

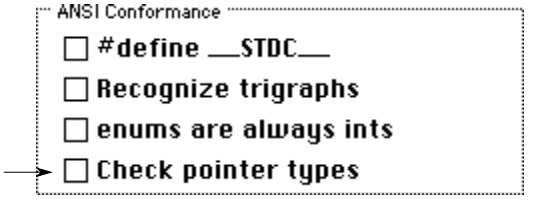

Figure 16-3 The Check pointer types option

When you're precompiling a header file, turning this option off can do something else. If this option is off and the Require prototypes option is on, the pointer types in the argument lists of the header's prototypes are changed to `void *`. For example, it's as if this prototype:

```
WindowPtr GetNewWindow( int windowID,
   Ptr wStorage, WindowPtr behind );
```

were defined like this:

```
WindowPtr GetNewWindow( int windowID,
   void *wStorage, void *behind );
```

Notice that the return type of the function isn't changed.

In a precompiled header, you set options with the `#pragma options` directive, as described in "Accessing Option Settings in Your Code" in Chapter 15. To turn off the Check pointer types option and turn on the Require prototypes option, include this line in your precompiled header:

```
#pragma options(!check_ptrs,
   require_protos)
```

Note

When you're precompiling `MacHeaders`, don't change the Check pointer types option. Instead, change the definition of the `SIMPLIFY_PROTOTYPES` macro. See "Precompiling Headers" on page 212 for more information.

To relax prototype checking when you precompile a header file, see "Checking pointer types" earlier in this chapter.

Enforcing prototype use

The Strict Prototype Enforcement option lets you choose how strictly THINK C enforces the use of prototypes. If this option is on, you can choose between two enforcement levels: Infer prototypes and Require prototypes. If this option is off, THINK C does nothing when you use or define a function without a prototype.

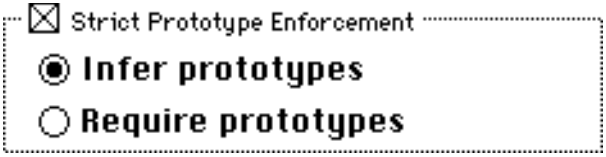

Figure 16-4 The Strict Prototype Enforcement option

This table explains the two enforcement levels:

If you choose…	When you use a function without a prototype, THINK C…
Infer prototypes	Infers a prototype from the first appearance of a function. That appearance can be a function call or an old-style declaration. If a subsequent call, declaration, or prototype doesn't match the inferred prototype, it's an error.
Require prototypes	Raises an error. You can't use or define a function unless it has a prototype. New-style function definitions do not satisfy this requirement.

Table 16-3 Prototype enforcement levels

Note

There is one exception to the Require prototypes option requirement. A static function does not need a prototype. Just define it before using it.

When you use the Require prototypes option, THINK C does not perform argument promotion on an old-style function definition, if there is a prototype for it. Take this prototype and old-style definition for the same function:

```
int AFunction(char);

AFunction(b)
   char b;
{
   ...
}
```

If Require prototypes is not selected, THINK C promotes `b` to an `int`, and the prototype and definition don't match. If Require prototypes is selected, THINK C treats the old-style definition as a new-style definition, and the prototype and definition match.

These examples show you how THINK C infers prototypes. If the first appearance of `PrintFloat()` is this call:

```
PrintFloat("Pi is ", 3.14159);
```

THINK C infers this prototype:

```
int PrintFloat( char *, double );
```

And if the first appearance of `PrintInt()` is this old-style declaration:

```
PrintInt( string, value )
   char *string;
   int value;
{
   printf( "%s%d\n", string, value );
}
```

THINK C infers this prototype:

```
int PrintInt( char *, int );
```

THINK C follows the rules of argument promotion when it infers prototypes. For example, it promotes parameters of type `char` and `short` to `int`, and parameters of type `float` to `double`. For example, from this function call:

```
PrintShort("This answer is: ", (short) 32);
```

THINK C infers this prototype: (Note that THINK C promotes the `short int` to `int`, and that it infers that a function with no specified return type returns an `int`.)

```
int PrintChar( char *, int );
```

Compiling Processor-Specific Code

You can test and change the settings of these options from your code. See "Accessing Option Settings in Your Code" in Chapter 15.

THINK C lets you compile code for a specific processor. You can compile code for the MC68020 CPU or the MC68881 floating-point unit. To compile processor-specific code, use these three options on the Compiler Settings page of the **THINK C options** dialog, shown in Figure 16-5: Generate 68020 instructions, Generate 68881 instructions, and Native floating-point format.

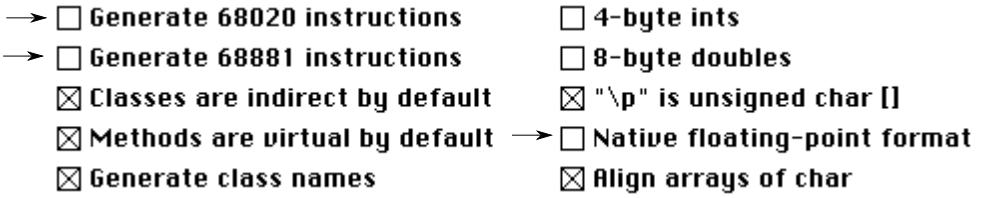

Figure 16-5 The options for compiling processor-specific code

Note

Before your program uses code compiled for a specific processor, make sure that processor is present, or your program may crash. Use the Gestalt Manager, described in *Inside Macintosh VI*, Chapter 3, "Compatibility Guidelines," or `SysEnvirons()`, described in *Inside Macintosh V*, Chapter 1, "Compatibility Guidelines."

Compiling code for the MC68020

To compile code for a Macintosh with a MC68020, MC68030, or MC68040, turn on the Generate 68020 instructions option. THINK C generates code that is optimized for Macintosh computers with the

MC68020, MC68030, or MC68040 (like the Macintosh LC, II, IIfx, or Quadra), and that will not run on Macintosh computers with the MC68000 (like the Macintosh Classic or Plus).

If the option is on, THINK C generates MC68020 instructions for bit-field operations, addressing, and long-word multiplication, division, and modulo operations. Also, the inline assembler accepts all MC68020 instructions and addressing modes for assembly in an `asm { … }` construct.

Compiling code for the MC68881

To compile code for a Macintosh with a MC68881, turn on the Generate MC68881 instructions option. THINK C generates code that is optimized for Macintosh computers with the MC68881 or MC68882 floating-point unit (like the Macintosh II or IIfx) or the MC68040 (like the Macintosh Quadra), and that will not run on Macintosh computers without it (like the Macintosh Classic, LC, or Plus).

For more information on using register variables, see "Using register variables" in Chapter 15.

If this option is on, THINK C generates code for the floating-point coprocessor. You may declare up to four local floating-point variables as `register` variables, and THINK C will place them into MC68881 registers.

For more information on how THINK C represents floating-point numbers, see "Floating-Point Representation" in Chapter 15.

If your code contains a lot of intensive floating-point computation, you may want to turn on the Native floating-point option. If both the Generate MC68881 instructions option and the Native floating-point format option are on, `long double` types use the MC68881 Extended format. If you call any SANE functions, you'll have to convert from the MC68881 Extended format and the SANE Extended format. The `float` and `short double` types are identical for SANE and for IEEE.

Note

If you turn on the Native floating-point option, you need to recompile the ANSI library if your project uses it.

Compiling Code to Debug It

*To turn on the THINK Debugger, choose **Use Debugger** from the **Project** menu. For more information, see Chapter 12, "Debugging Programs."*

THINK C has several options to make debugging your code easier. The options on the Debugging page of the **THINK C options** dialog

let you add information to your compiled code that makes it easier to use debuggers, like the THINK Debugger or MacsBug.

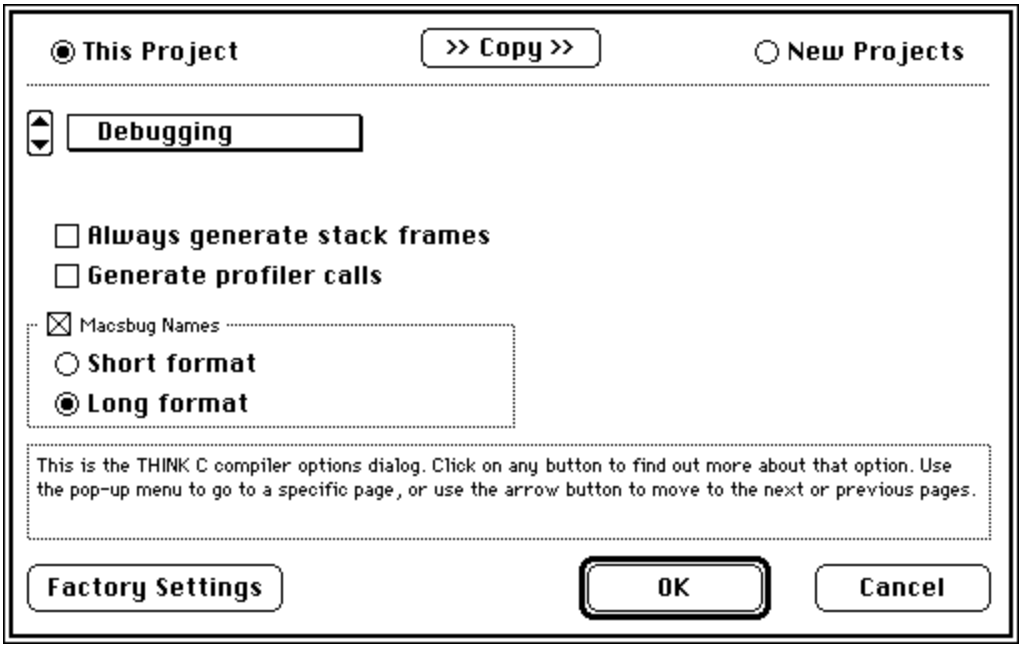

Figure 16-6 The Debugging page of the THINK C options dialog

Always generate stack frames

If this option is on, THINK C always generates a stack frame (`LINK`/`UNLK` instructions), even for functions that normally wouldn't have one. If a function has no local variables and takes no parameters, THINK C does not generate a stack frame for it unless this option is on. Since the call chain pop-up menu relies on stack frames, you may want to turn on this option while you're debugging.

Note

If a function has no local variables, takes no parameters, and uses inline assembly, THINK C will not generate a stack frame for it, even if this option is on.

Generate profiler calls

When this option is on, THINK C generates code for the profiler. To learn about the profiler, see Chapter 25, “Profiling Projects.” This option doesn’t affect the operation of the THINK Debugger.

MacsBug Names

If this option is on, THINK C generates names for low-level debuggers like MacsBug so they can find the name of a function. There are two varieties of MacsBug names:

Format	Description
Short format	Uses only the first eight characters of the function name. All characters are translated to uppercase.
Long format	Uses all the characters of the function name. Case is preserved.

Table 16-4 MacsBug Names option

THINK C generates MacsBug symbols only for functions that have stack frames. If you have functions in your program that normally wouldn’t require a stack frame, you might want to turn on the Always generate stack frames options.

Both kinds of MacsBug names appear right after the code for a function, so turning this option on will increase the size of your code. You should turn this option off when you build the final version of your application, desk accessory, device driver, or code resource.

Using the THINK C Project Prefix

The THINK C project prefix lets you include some text in all the C source files in your project. It’s as if you put the project prefix at the

beginning of all your C source files. You set this prefix in the Prefix page of the **THINK C options** dialog, shown in Figure 16-7.

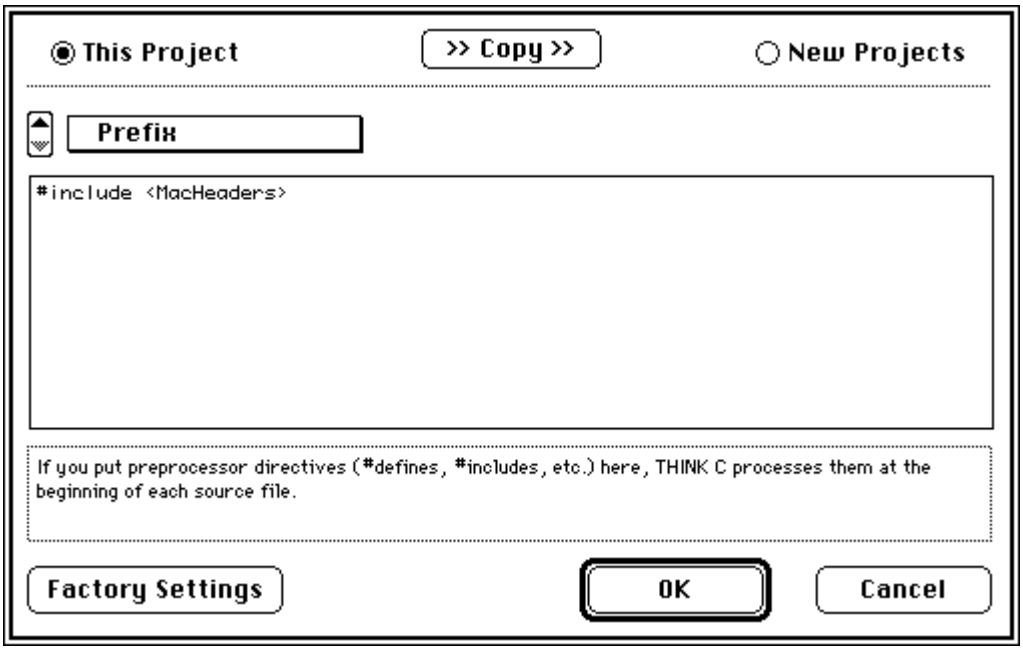

Figure 16-7 The Prefix page of the THINK C options dialog

THINK C, Symantec C++, and THINK Rez keep separate project prefixes. If your project contains C, C++, and Rez source files and you want the same project prefix for all the source files, you need to enter the prefix *three times*: in the THINK C Prefix page, the Symantec C++ Prefix page, and the THINK Rez Prefix string box.

If you use a precompiled header file, like `MacHeaders`, include it here. By default, this page contains the line `#include <MacHeaders>`.

Note

THINK C does not precompile what's in the project prefix. To precompile something, you must use the **Precompile** command described in Chapter 11, "Running Programs."

If you need to define a macro in all your files, define it here. For example, you may have some debugging code in your files that's compiled only if the macro `DEBUG` is defined. To include that code, include this line here:

```
#define DEBUG
```

When you don't need to include the debugging code anymore, just delete that line from this page. You don't need to edit every file in your project.

Note

When you change the THINK C project prefix, the THINK Project Manager marks all the THINK C source files in your project for recompilation.